CS 423
Computer Architecture
Spring 2012

Lecture 06:
VLIW Processors

Ozcan Ozturk

http://www.cs.bilkent.edu.tr/~ozturk/cs423/

[Adapted from *Computer Organization and Design*,
Patterson & Hennessy, © 2005, UCB]

Review: Multi-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software
  - Data dependencies – aka data hazards
    - True data dependencies (read after write)
      - Use data forwarding hardware
      - Use compiler scheduling
    - Storage dependence (aka name dependence)
      - Use register renaming to solve both
        - » Antidependencies (write after read)
        - » Output dependencies (write after write)
  - Procedural dependencies – aka control hazards
    - Use aggressive branch prediction (speculation)
    - Use predication
  - Resource conflicts – aka structural hazards
    - Use resource duplication or resource pipelining to reduce (or eliminate) resource conflicts
    - Use arbitration for result and commit buses and register file read and write ports

Review: Multiple-Issue Processor Styles

- Dynamic multiple-issue processors (aka superscalar)
  - Decisions on which instructions to execute simultaneously (in the range of 2 to 8 in 2005) are being made dynamically (at run time by the hardware)
  - E.g., IBM Power 2, Pentium 4, MIPS R10K, HP PA 8500 IBM

- Static multiple-issue processors (aka VLIW)
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
  - E.g., Intel Itanium and Itanium 2 for the IA-64 ISA – EPIC (Explicit Parallel Instruction Computer)
    - 128 bit "bundles" containing 3 instructions each 41 bits + 5 bit template field (specifies which FU each instr needs)
    - Five functional units (IntALU, MMedia, DMem, FPALU, Branch)
    - Extensive support for speculation and predication

History of VLIW Processors

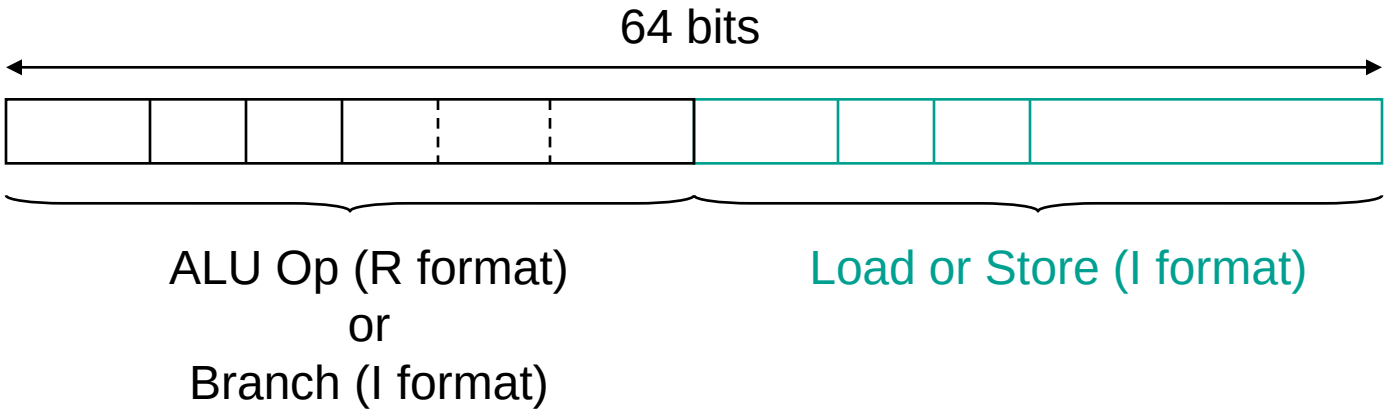
- Started with (horizontal) microprogramming
  - Very wide microinstructions used to directly generate control signals in single-issue processors (e.g., IBM 360 series)
- VLIW for multi-issue processors first appeared in the Multiflow and Cydrome (in the early 1980's)
- Current commercial VLIW processors
  - Intel i860 RISC (dual mode: scalar and VLIW)
  - Intel I-64 (EPIC: Itanium and Itanium 2)
  - Transmeta Crusoe
  - Lucent/Motorola StarCore
  - ADI TigerSHARC
  - Infineon (Siemens) Carmel

Static Multiple Issue Machines (VLIW)

- Static multiple-issue processors (aka VLIW) use the compiler to decide which instructions to issue and execute simultaneously
  - Issue packet – the set of instructions that are bundled together and issued in one clock cycle – think of it as one large instruction with multiple operations
  - The mix of instructions in the packet (bundle) is usually restricted – a single "instruction" with several predefined fields
  - The compiler does static branch prediction and code scheduling to reduce (control) or eliminate (data) hazards
- VLIW's have
  - Multiple functional units (like SS processors)
  - Multi-ported register files (again like SS processors)
  - Wide program bus

An Example: A VLIW MIPS

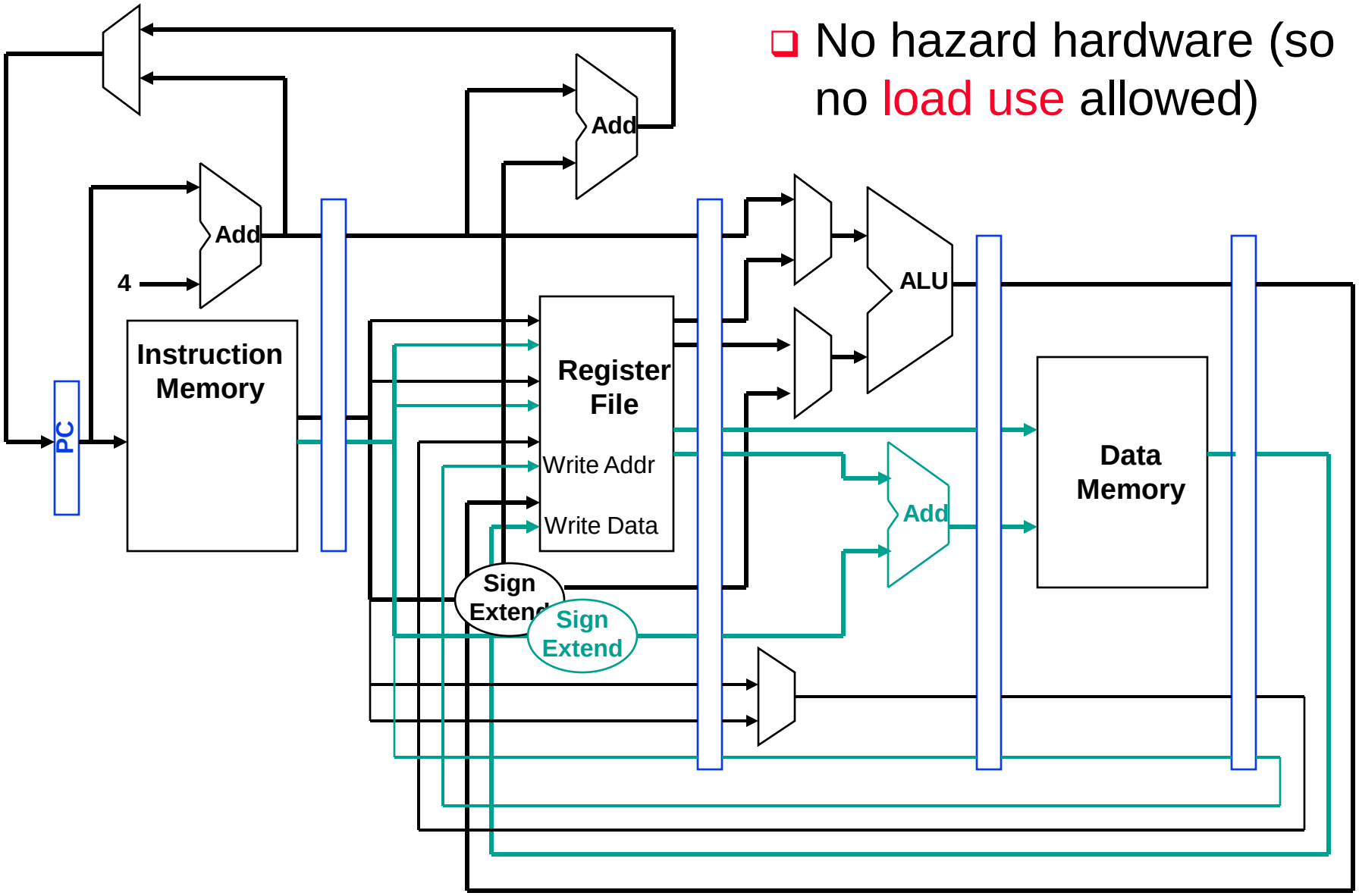
- Consider a 2-issue MIPS with a 2 instr bundle



- Instructions are always fetched, decoded, and issued in pairs
  - If one instr of the pair can not be used, it is replaced with a noop
- Need 4 read ports and 2 write ports and a separate memory address adder

A MIPS VLIW (2-issue) Datapath

- No hazard hardware (so no load use allowed)



Code Scheduling Example

- Consider the following loop code

```
lp:   lw    $t0,0($s1)    # $t0=array element
      addu  $t0,$t0,$s2   # add scalar in $s2
      sw    $t0,0($s1)    # store result
      addi  $s1,$s1,-4    # decrement pointer
      bne   $s1,$0,lp     # branch if $s1 != 0
```

- Must "schedule" the instructions to avoid pipeline stalls
  - Instructions in one bundle *must* be independent
  - Must separate load use instructions from their loads by one cycle
  - Notice that the first two instructions have a load use dependency, the next two and last two have data dependencies
  - Assume branches are perfectly predicted by the hardware

The Scheduled Code (Not Unrolled)

	ALU or branch	Data transfer	CC
lp:		lw $t0,0($s1)	1
	addi $s1,$s1,-4		2
	addu $t0,$t0,$s2		3
	bne $s1,$0,lp	sw $t0,4($s1)	4

- Four clock cycles to execute 5 instructions for a
  - CPI of 0.8 (versus the best case of 0.5)
  - IPC of 1.25 (versus the best case of 2.0)
  - noops don't count towards performance !!

Loop Unrolling

- Loop unrolling – multiple copies of the loop body are made and instructions from different iterations are scheduled together as a way to increase ILP

- Apply loop unrolling (4 times for our example) and then schedule the resulting code
  - Eliminate unnecessary loop overhead instructions
  - Schedule so as to avoid load use hazards

- During unrolling the compiler applies register renaming to eliminate all data dependencies that are not true dependencies

Unrolled Code Example

```
lp:   lw    $t0,0($s1)      # $t0=array element
      lw    $t1,-4($s1)     # $t1=array element
      lw    $t2,-8($s1)     # $t2=array element
      lw    $t3,-12($s1)    # $t3=array element
      addu  $t0,$t0,$s2     # add scalar in $s2
      addu  $t1,$t1,$s2     # add scalar in $s2
      addu  $t2,$t2,$s2     # add scalar in $s2
      addu  $t3,$t3,$s2     # add scalar in $s2
      sw    $t0,0($s1)      # store result
      sw    $t1,-4($s1)     # store result
      sw    $t2,-8($s1)     # store result
      sw    $t3,-12($s1)    # store result
      addi  $s1,$s1,-16     # decrement pointer
      bne   $s1,$0,lp       # branch if $s1 != 0
```

The Scheduled Code (Unrolled)

	ALU or branch	Data transfer	CC
lp:	addi $s1,$s1,-16	lw $t0,0($s1)	1
		lw $t1,12($s1)	2
	addu $t0,$t0,$s2	lw $t2,8($s1)	3
	addu $t1,$t1,$s2	lw $t3,4($s1)	4
	addu $t2,$t2,$s2	sw $t0,16($s1)	5
	addu $t3,$t3,$s2	sw $t1,12($s1)	6
		sw $t2,8($s1)	7
	bne $s1,$0,lp	sw $t3,4($s1)	8

- Eight clock cycles to execute 14 instructions for a
  - CPI of 0.57 (versus the best case of 0.5)
  - IPC of 1.8 (versus the best case of 2.0)

Speculation

- Speculation is used to allow execution of future instr's that (may) depend on the speculated instruction
  - Speculate on the outcome of a conditional branch (branch prediction)
  - Speculate that a store (for which we don't yet know the address) that precedes a load does not refer to the same address, allowing the load to be scheduled before the store (load speculation)
- Must have (hardware and/or software) mechanisms for
  - Checking to see if the guess was correct
  - Recovering from the effects of the instructions that were executed speculatively if the guess was incorrect
    - In a VLIW processor the compiler can insert additional instr's that check the accuracy of the speculation and can provide a fix-up routine to use when the speculation was incorrect
- Ignore and/or buffer exceptions created by speculatively executed instructions until it is clear that they should really occur

Predication

- Predication can be used to eliminate branches by making the execution of an instruction dependent on a "predicate", e.g.,

```
if (p) {statement 1} else {statement 2}
```

  would normally compile using two branches. With predication it would compile as

```
(p) statement 1
(~p) statement 2
```

- The use of `(condition)` indicates that the instruction is committed only if `condition` is true
- Predication can be used to speculate as well as to eliminate branches

Compiler Support for VLIW Processors

- The compiler packs groups of independent instructions into the bundle
  - Done by code re-ordering (trace scheduling)
- The compiler uses loop unrolling to expose more ILP
- The compiler uses register renaming to solve name dependencies and ensures no load use hazards occur
- While superscalars use dynamic prediction, VLIW's primarily depend on the compiler for branch prediction
  - Loop unrolling reduces the number of conditional branches
  - Predication eliminates if-the-else branch structures by replacing them with predicated instructions
- The compiler predicts memory bank references to help minimize memory bank conflicts

CISC vs RISC vs SS vs VLIW

	CISC	RISC	Superscalar	VLIW
Instr size	variable size	fixed size	fixed size	fixed size (but large)
Instr format	variable format	fixed format	fixed format	fixed format
Registers	few, some special	many GP	GP and rename (RUU)	many, many GP
Memory reference	embedded in many instr's	load/store	load/store	load/store
Key Issues	decode complexity	data forwarding, hazards	hardware dependency resolution	(compiler) code scheduling
Instruction flow